Republic of the Philippines

POLYTECHNIC UNIVERSITY OF THE PHILIPPINES

College of Engineering

Sta. Mesa, Manila

Data Structures and Algorithm

Group Laboratory Exercise - Sorting Algorithms

Group 1

In partial fulfillment of the requirements in

CMPE 201 Data Structures and Algorithms

Academic Year 2023–2024

Submitted by:

Baay, Frances Leigh G.

Legaspi, Ken Andrea G.

Mallari, Ella Marie A.

Membrere, Zemerelin Iris M.

Pabuna, Katrina B.

Sajo, Jhon Kaiser R.

Sevial, Rebekah Joy E.

Submitted to:

Prof. Godofredo T. Avena

February 3, 2024

Data Structures and Algorithm

Group Laboratory Exercise - Sort Algorithms

Group 1

Objective:

Research on the python implementation and add it to you group work as final Requirements. Also explain the big O complexity for each sorting Algorithm.

Sorting Algorithms:

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quicksort

Introduction

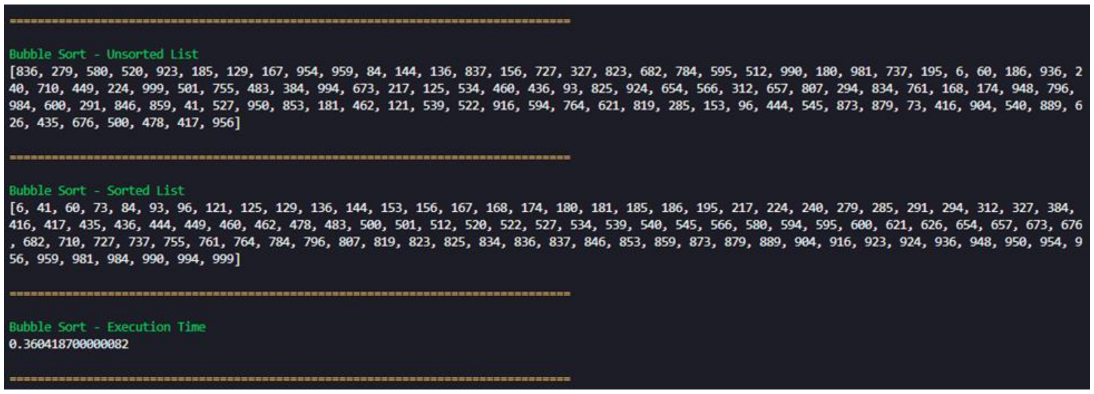
An array or list of elements can be rearranged using a sorting algorithm based on a comparison operator applied to the elements. The operator for comparison determines the new element order in the corresponding data structure. There are different types of sorting algorithms however, in this paper, only some will be discussed. Five of the many types of sorting algorithms such as bubble sort, insertion sort, selection sort, merge sort, and quick sort will be explained in detail.

These sorting algorithms have different time complexities and performance characteristics, making them suitable for different scenarios. Understanding how each algorithm works and when to use them can greatly improve the efficiency of sorting operations in various applications. Additionally, exploring the advantages and disadvantages of these algorithms can help in selecting the most appropriate one for a specific use case.

PERFORMANCE OF THE SORTING ALGORITHMS

Figure 1

Bubble Sort



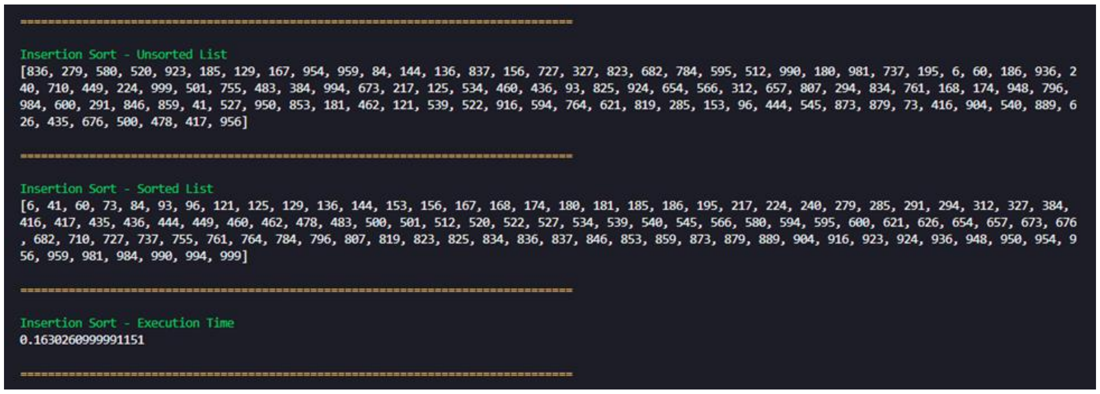
```
================================================================================

Bubble Sort - Unsorted List
[836, 279, 580, 520, 923, 185, 129, 167, 954, 959, 84, 144, 136, 837, 156, 727, 327, 823, 682, 784, 595, 512, 990, 180, 981, 737, 195, 6, 60, 186, 936, 2
40, 710, 449, 224, 999, 501, 755, 483, 384, 994, 673, 217, 125, 534, 460, 436, 93, 825, 924, 654, 566, 312, 657, 807, 294, 834, 761, 168, 174, 948, 796,
984, 600, 291, 846, 859, 41, 527, 950, 853, 181, 462, 121, 539, 522, 916, 594, 764, 621, 819, 285, 153, 96, 444, 545, 873, 879, 73, 416, 904, 540, 889, 6
26, 435, 676, 500, 478, 417, 956]

================================================================================

Bubble Sort - Sorted List
[6, 41, 60, 73, 84, 93, 96, 121, 125, 129, 136, 144, 153, 156, 167, 168, 174, 180, 181, 185, 186, 195, 217, 224, 240, 279, 285, 291, 294, 312, 327, 384,
416, 417, 435, 436, 444, 449, 460, 462, 478, 483, 500, 501, 512, 520, 522, 527, 534, 539, 540, 545, 566, 580, 594, 595, 600, 621, 626, 654, 657, 673, 676
, 682, 710, 727, 737, 755, 761, 764, 784, 796, 807, 819, 823, 825, 834, 836, 837, 846, 853, 859, 873, 879, 889, 904, 916, 923, 924, 936, 948, 950, 954, 9
56, 959, 981, 984, 990, 994, 999]

================================================================================

Bubble Sort - Execution Time
0.360418700000082

================================================================================
```

Note. Both Sorted and Unsorted lists consist of 100 random integers. The code can also be found on the repository of where this file is located.

The bubble algorithm is known for its simplicity and effectiveness in sorting. It operates by continuously comparing and swapping adjacent elements until the entire list is sorted. In Figure 1, the data clearly demonstrates that this algorithm performs consistently well, with an average execution time of 0.360418 milliseconds for both sorted and unsorted lists.

Figure 2

Insertion Sort



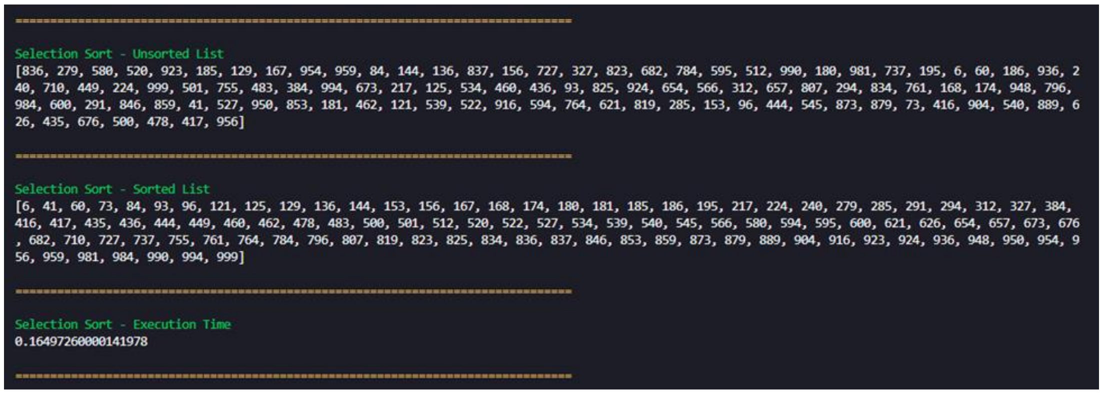
```
================================================================================

Insertion Sort - Unsorted List
[836, 279, 580, 520, 923, 185, 129, 167, 954, 959, 84, 144, 136, 837, 156, 727, 327, 823, 682, 784, 595, 512, 990, 180, 981, 737, 195, 6, 60, 186, 936, 2
40, 710, 449, 224, 999, 501, 755, 483, 384, 994, 673, 217, 125, 534, 460, 436, 93, 825, 924, 654, 566, 312, 657, 807, 294, 834, 761, 168, 174, 948, 796,
984, 600, 291, 846, 859, 41, 527, 950, 853, 181, 462, 121, 539, 522, 916, 594, 764, 621, 819, 285, 153, 96, 444, 545, 873, 879, 73, 416, 904, 540, 889, 6
26, 435, 676, 500, 478, 417, 956]

================================================================================

Insertion Sort - Sorted List
[6, 41, 60, 73, 84, 93, 96, 121, 125, 129, 136, 144, 153, 156, 167, 168, 174, 180, 181, 185, 186, 195, 217, 224, 240, 279, 285, 291, 294, 312, 327, 384,
416, 417, 435, 436, 444, 449, 460, 462, 478, 483, 500, 501, 512, 520, 522, 527, 534, 539, 540, 545, 566, 580, 594, 595, 600, 621, 626, 654, 657, 673, 676
, 682, 710, 727, 737, 755, 761, 764, 784, 796, 807, 819, 823, 825, 834, 836, 837, 846, 853, 859, 873, 879, 889, 904, 916, 923, 924, 936, 948, 950, 954, 9
56, 959, 981, 984, 990, 994, 999]

================================================================================

Insertion Sort - Execution Time
0.1630260999991151

================================================================================
```

Note. Both Sorted and Unsorted lists consist of 100 random integers. The code can also be found on the repository of where this file is located.

The insertion algorithm, on the other hand, functions by splitting the array into sorted and unsorted parts. It is also known for its efficiency when sorting small arrays or partially sorted arrays. To sort an array of size N in ascending order, it iterates over the array and compares the current element (key) to its predecessor; if the key element is smaller than its predecessor, it compares it to the elements before. Move the greater elements one position up to make space for the swapped element. In Figure 2, this algorithm gives an average of 0.1630261 milliseconds on both sorted and unsorted lists, which consist of 100 random integers. It performs well in these scenarios because it only needs to make a few comparisons and swaps to sort the elements.

Figure 3

Selection Sort



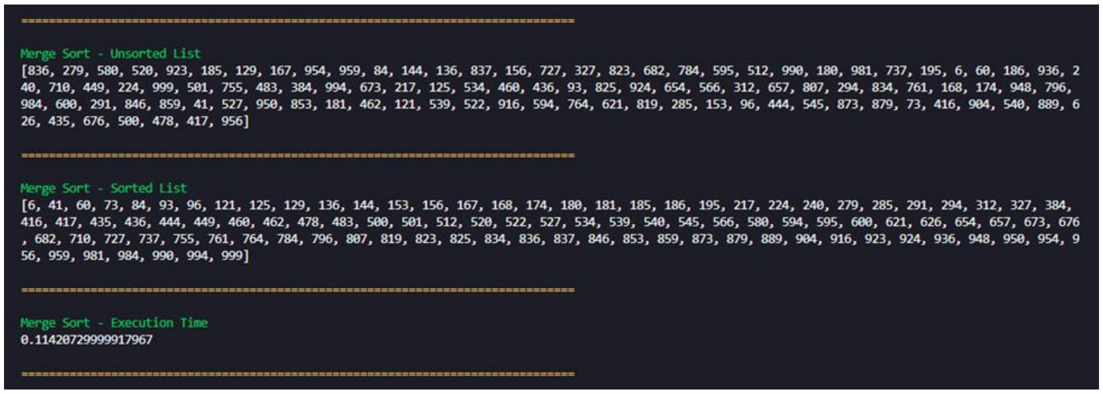
```
================================================================================

Selection Sort - Unsorted List
[836, 279, 580, 520, 923, 185, 129, 167, 954, 959, 84, 144, 136, 837, 156, 727, 327, 823, 682, 784, 595, 512, 990, 180, 981, 737, 195, 6, 60, 186, 936, 2
40, 710, 449, 224, 999, 501, 755, 483, 384, 994, 673, 217, 125, 534, 460, 436, 93, 825, 924, 654, 566, 312, 657, 807, 294, 834, 761, 168, 174, 948, 796,
984, 600, 291, 846, 859, 41, 527, 950, 853, 181, 462, 121, 539, 522, 916, 594, 764, 621, 819, 285, 153, 96, 444, 545, 873, 879, 73, 416, 904, 540, 889, 6
26, 435, 676, 500, 478, 417, 956]

================================================================================

Selection Sort - Sorted List
[6, 41, 60, 73, 84, 93, 96, 121, 125, 129, 136, 144, 153, 156, 167, 168, 174, 180, 181, 185, 186, 195, 217, 224, 240, 279, 285, 291, 294, 312, 327, 384,
416, 417, 435, 436, 444, 449, 460, 462, 478, 483, 500, 501, 512, 520, 522, 527, 534, 539, 540, 545, 566, 580, 594, 595, 600, 621, 626, 654, 657, 673, 676
, 682, 710, 727, 737, 755, 761, 764, 784, 796, 807, 819, 823, 825, 834, 836, 837, 846, 853, 859, 873, 879, 889, 904, 916, 923, 924, 936, 948, 950, 954, 9
56, 959, 981, 984, 990, 994, 999]

================================================================================

Selection Sort - Execution Time
0.16497260000141978

================================================================================
```

Note. Both Sorted and Unsorted lists consist of 100 random integers. The code can also be found on the repository of where this file is located.

Another sorting algorithm is the selection sort. This algorithm is also a simple yet efficient sorting algorithm that works by repeating the process of selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The data in Figure 3 shows that using 100 random integers, the average execution time for this sorting algorithm for both sorted and unsorted lists is 0.1649726 milliseconds. This indicates that the sorting algorithm has consistent and efficient performance regardless of the initial ordering of the list.

Figure 4

Merge Sort



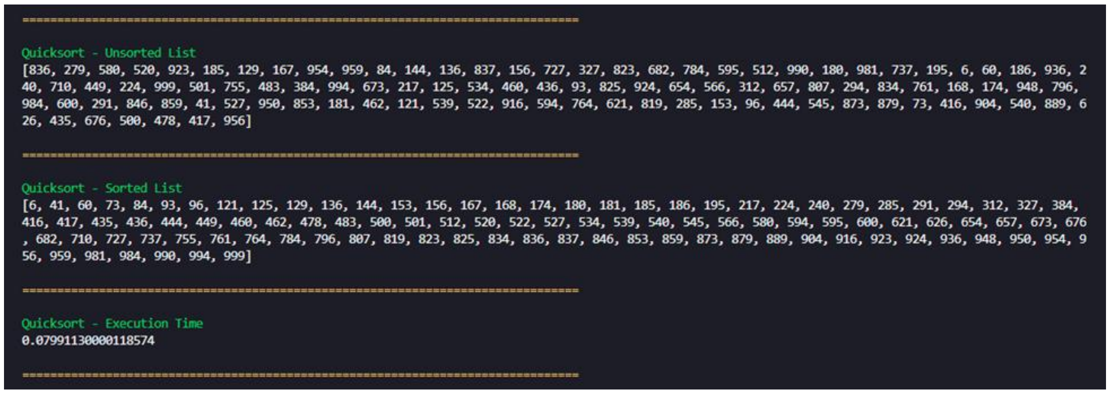
```
================================================================================

Merge Sort - Unsorted List
[836, 279, 580, 520, 923, 185, 129, 167, 954, 959, 84, 144, 136, 837, 156, 727, 327, 823, 682, 784, 595, 512, 990, 180, 981, 737, 195, 6, 60, 186, 936, 2
40, 710, 449, 224, 999, 501, 755, 483, 384, 994, 673, 217, 125, 534, 460, 436, 93, 825, 924, 654, 566, 312, 657, 807, 294, 834, 761, 168, 174, 948, 796,
984, 600, 291, 846, 859, 41, 527, 950, 853, 181, 462, 121, 539, 522, 916, 594, 764, 621, 819, 285, 153, 96, 444, 545, 873, 879, 73, 416, 904, 540, 889, 6
26, 435, 676, 500, 478, 417, 956]

================================================================================

Merge Sort - Sorted List
[6, 41, 60, 73, 84, 93, 96, 121, 125, 129, 136, 144, 153, 156, 167, 168, 174, 180, 181, 185, 186, 195, 217, 224, 240, 279, 285, 291, 294, 312, 327, 384,
416, 417, 435, 436, 444, 449, 460, 462, 478, 483, 500, 501, 512, 520, 522, 527, 534, 539, 540, 545, 566, 580, 594, 595, 600, 621, 626, 654, 657, 673, 676
, 682, 710, 727, 737, 755, 761, 764, 784, 796, 807, 819, 823, 825, 834, 836, 837, 846, 853, 859, 873, 879, 889, 904, 916, 923, 924, 936, 948, 950, 954, 9
56, 959, 981, 984, 990, 994, 999]

================================================================================

Merge Sort - Execution Time
0.11420729999917967

================================================================================
```

Note. Both Sorted and Unsorted lists consist of 100 random integers. The code can also be found on the repository of where this file is located.

The merge sorting method is another option. It functions by first splitting larger arrays into smaller subarrays, sorting each subarray separately, and then combining the sorted subarrays back into the original array to create the final sorted array. This can alternatively be thought of as a recurrence of the procedure, where the array is divided in half, each half is sorted, and then the sorted halves are merged back together. The data in Figure 4 illustrates the 100 random numbers in each sorted and unsorted list, and the average execution time is 0.1140273 milliseconds. The data further illustrates the sorting algorithm's efficacy and efficiency.

Figure 5

Quick Sort



```
================================================================================

Quicksort - Unsorted List
[836, 279, 580, 520, 923, 185, 129, 167, 954, 959, 84, 144, 136, 837, 156, 727, 327, 823, 682, 784, 595, 512, 990, 180, 981, 737, 195, 6, 60, 186, 936, 2
40, 710, 449, 224, 999, 501, 755, 483, 384, 994, 673, 217, 125, 534, 460, 436, 93, 825, 924, 654, 566, 312, 657, 807, 294, 834, 761, 168, 174, 948, 796,
984, 600, 291, 846, 859, 41, 527, 950, 853, 181, 462, 121, 539, 522, 916, 594, 764, 621, 819, 285, 153, 96, 444, 545, 873, 879, 73, 416, 904, 540, 889, 6
26, 435, 676, 500, 478, 417, 956]

================================================================================

Quicksort - Sorted List
[6, 41, 60, 73, 84, 93, 96, 121, 125, 129, 136, 144, 153, 156, 167, 168, 174, 180, 181, 185, 186, 195, 217, 224, 240, 279, 285, 291, 294, 312, 327, 384,
416, 417, 435, 436, 444, 449, 460, 462, 478, 483, 500, 501, 512, 520, 522, 527, 534, 539, 540, 545, 566, 580, 594, 595, 600, 621, 626, 654, 657, 673, 676
, 682, 710, 727, 737, 755, 761, 764, 784, 796, 807, 819, 823, 825, 834, 836, 837, 846, 853, 859, 873, 879, 889, 904, 916, 923, 924, 936, 948, 950, 954, 9
56, 959, 981, 984, 990, 994, 999]

================================================================================

Quicksort - Execution Time
0.07991130000118574

================================================================================
```

Note. Both Sorted and Unsorted lists consist of 100 random integers. The code can also be found on the repository of where this file is located.

And lastly, the quick sort. Choosing an element to serve as a pivot and dividing the input array around it by positioning the pivot in the proper location inside the sorted array is how this sorting method works. This algorithm was developed as an extension of the Divide and Conquer Algorithm. The results shown in Figure 5 demonstrate that, for each sorted and unsorted list containing 100 random integers, the average execution time is 0.0799113 milliseconds, indicating a rather fast function.

RESULTS AND DISCUSSION

Table 1

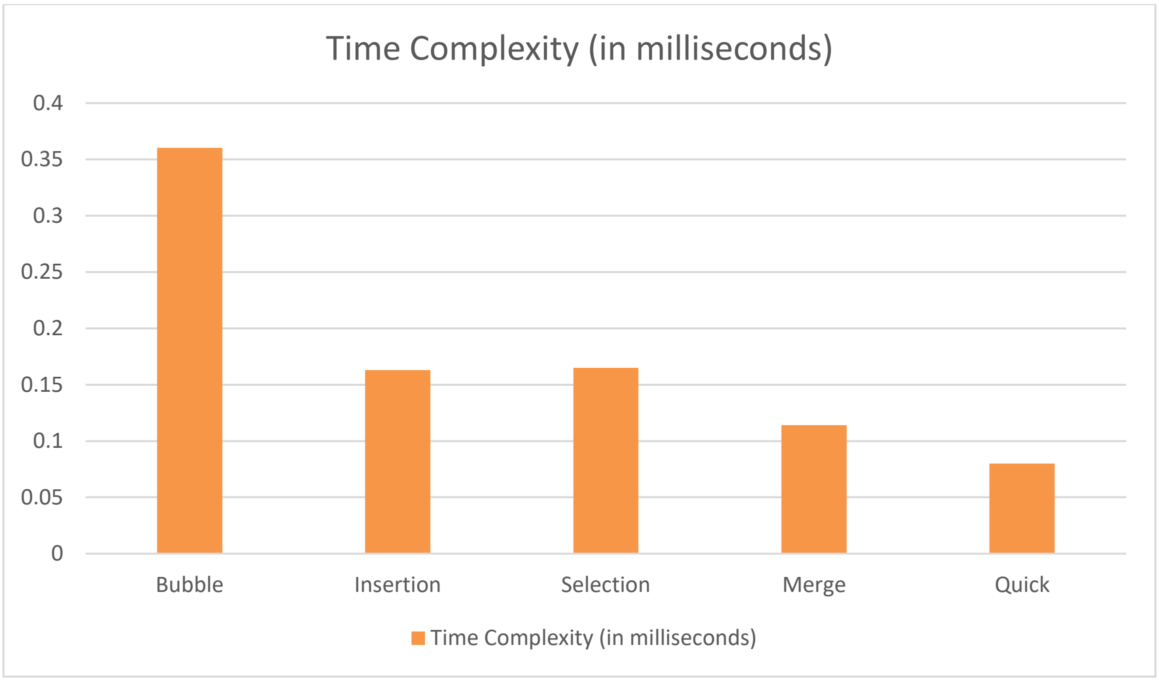
Sorting Algorithms Analysis

Execution Time for Each Sorting Algorithms (in milliseconds)	
Bubble	0.360418ms
Insertion	0.1630261ms
Selection	0.1649726ms
Merge	0.1140273ms
Quick	0.0799113ms

By analyzing the execution time for each sorting algorithm in table 1, it shows that bubble sort does its function in 0.360418 milliseconds, which is the slowest among all the sorting algorithms mentioned. However, quick sort performs the fastest, with an execution time of 0.0799113 milliseconds. With the other algorithms, such as insertion sort, selection sort, and merge sort, the execution times are 0.1649726 milliseconds, 0.1649726 milliseconds, and 0.1140273 milliseconds, respectively. This indicates the importance of considering the time complexity and efficiency of different sorting algorithms when choosing the most appropriate one for a particular task.

Figure 6

Sorting Algorithms Time Complexity Graph



The time complexity of a sorting algorithm refers to the amount of time it takes to execute the algorithm as the size of the input increases. In table 1 and figure 6, bubble sort, insertion sort, and selection sort have a quadratic time complexity, O(n^2), because their execution time grows exponentially with the size of the input. On the other hand, merge sort and quick sort have a logarithmic time complexity, O (n log n), as their execution time grows at a slower rate compared to the size of the input.

O(n^2) executes slower and less efficient than O (n log n) because O(n^2) uses nested loops or repeated comparisons which lead to a quadratic increase in the number of operations as the input size increases. While, O (n log n) divides the input data into smaller portions recursively and then merge or combine them efficiently, resulting in a more efficient algorithm overall. This makes O (n log n) a better choice for larger input sizes.